Large and Dynamic Variables/Fields

The following topics are covered below:

Introduction

Beginning with Natural for Windows and UNIX/OpenVMS Version 4.1, Natural enhanced capabilities are provided for the usage of large variables by removing the existing size limitations and by providing for dynamic allocation of these variables at execution time.

Large variables for alphanumeric and binary data are based on the well known Natural formats A and B. The limitations of 253 for Format A and 126 for Format B are no longer in effect. The new size limit is 1 GB. These large static variables and fields are handled in the same manner as traditional alphanumeric and binary variables and fields with regard to definition, redefinition, value space allocation, conversions, referencing in statements, etc. All rules concerning alphanumeric and binary formats apply to these large formats.

In that the maximum size of large data structures (for example, pictures, sounds, videos) may not exactly be known at application development time, Natural additionally provides for the definition of alphanumeric and binary variables with the attribute DYNAMIC. The value space of variables which are defined with this attribute will be extended dynamically at execution time when it becomes necessary (for example, during an assignment operation: #picture1 := #picture2). This means that large binary and alphanumeric data structures may be processed in Natural without the need to define a limit at development time. The execution-time allocation of dynamic variables is of course subject to available memory restrictions. If the allocation of dynamic variables results in an insufficent memory condition being returned by the underlying operating system, the ON ERROR statement can be used to intercept this error condition; otherwise, an error message will be returned by Natural.
The Natural system variable *LENGTH can be used to obtain the number of bytes of the value space which are currently used for a given dynamic variable. Natural automatically sets *LENGTH to the length of the source operand during assignments in which the dynamic variable is involved. *LENGTH(field) therefore returns the size currently used for a dynamic Natural field or variable in bytes.

If the dynamic variable space is no longer needed, the REDUCE DYNAMIC VARIABLE statement can be used to reduce the space used for the dynamic variable to zero (or any other desired size). If the upper limit of memory usage is known for a specific dynamic variable, the EXPAND statement can be used to set the space used for the dynamic variable to this specific size.

If a dynamic variable is to be initialized, the MOVE ALL UNTIL statement should be used for this purpose.

Note concerning Natural RPC:
Large and dynamic variables are currently not supported by the Natural remote procedure call (RPC).

Definition of Dynamic Variables

Because the actual size of large alphanumeric and binary data structures may not be exactly known at application development time, the definition of ***dynamic*** variables of Format A or B can be used to manage these structures. The dynamic allocation and extension (reallocation) of large variables is transparent to the application programming logic. Dynamic variables are defined without any length. Memory allocation will be done at execution time

implicitly, when the dynamic variable is used as a target operand or explicitly with an EXPAND statement.

Dynamic variables can only be defined in a DEFINE DATA statement using the following syntax:

```
level   variable-name   ( A )    DYNAMIC
level   variable-name   ( B )    DYNAMIC
```

A dynamic variable can only be defined as **scalar** (no dynamic array definition is possible).
A dynamic variable may not be contained in a **REDEFINE clause**, and a **redefinition** of a dynamic variable or of a group that contains a dynamic variable is not possible. The **CONST** and **INIT** clauses are invalid for dynamic variables.

System Variable *LENGTH(field)

The size of the currently used value space of a dynamic variable can be obtained from the system variable *LENGTH. *LENGTH is set to the (used) length of the source operand during assignments automatically.

Note:
Due to performance considerations, the allocated size may be larger than the used size. It is not possible for the Natural programmer to obtain information about the currently allocated size. This is an internal value.

*LENGTH(field) returns the used size of a dynamic Natural field or variable in bytes. *LENGTH may be used only to get the currently used size for dynamic variables.

Size Limitation Checks

Profile Parameter DSLM

For compatibility reasons, a size limitation check at compile time for fixed length variables can be done using the DSLM parameter. The DSLM parameter limits the size to 32 KB per variable.

Profile Parameter USIZE

For dynamic variables, a size limitation check at compile time is not possible because no length is defined for dynamic variables. The Size of User Buffer Area (USIZE) indicates the size of the user buffer in virtual memory. The user buffer contains all data dynamically allocated by Natural. If a dynamic variable is allocated or extended at execution time and the USIZE limitation is exceeded, an error message will be returned.

Statements EXPAND and REDUCE

The statements EXPAND and REDUCE are used to explicitly allocate and free memory space for a dynamic variable.

Syntax:

```
EXPAND [ SIZE OF ] DYNAMIC [ VARIABLE ]  operand1  TO operand2
REDUCE [ SIZE OF ] DYNAMIC [ VARIABLE ]  operand1  TO operand2
```

where operand1 is a dynamic variable and operand2 is a non-negative numeric size value.

The EXPAND statement is used to extend the allocated size of the dynamic variable to a given size. The size currently used (*LENGTH) for the dynamic variable is not modified.
If the given size is less than the currently allocated size of the dynamic variable, the statement will be ignored.

The REDUCE statement is used to reduce the allocated memory size. The allocated memory of the dynamic variable beyond the given size is released immediately (when the statement is executed).
If the size currently used (*LENGTH) for the dynamic variable is greater than the given size, *LENGTH of this dynamic variable is set to this size. The content of the variable is truncated, but not modified. If the given size is larger than the currently allocated size of the dynamic variable, the statement will be ignored.

Usage of Dynamic Variables

- Assignments with Dynamic Variables
- Initialization of Dynamic Variables
- String Manipulation with Dynamic Alpha Variables
- Logical Condition Criterion (LCC) with Dynamic Variables
- Parameter Transfer with Dynamic Variables
- Work File Access with Large and Dynamic Variables
- DDM Generation and Editing for Varying Length Columns
- Accessing Large Database Objects
- Performance Aspects with Dynamic Variables

Generally, a dynamic alphanumeric variable may be used wherever an operand of Format A or Format B is allowed.

Exception:
Dynamic variables are not allowed within DISPLAY, WRITE, PRINT, STACK, INPUT, REINPUT, and SORT statements. (To DISPLAY, WRITE or PRINT dynamic alphanumeric variables, the variable must be cut into smaller portions using the MOVE SUBSTRING statement.)

The used length (*LENGTH) and the allocated size of dynamic variables are equal to zero until such time that the variable is first accessed as a target operand. Due to assignments or other manipulation operations, dynamic variables may be firstly allocated or extended (reallocated) to the exact size of the source operand.

The size of a dynamic variable may be extended if it is used as a modifiable operand (target operand) in the following statements:

- destination operand in an assignment (ASSIGN, MOVE)
- operand2 in COMPRESS
- operand1 in EXAMINE REPLACE
- operand4 in SEPARATE
- READ WORK FILE
- parameter or view field in the INTO clause of SELECT
- CALLNAT, PERFORM (except AD=O, or BY VALUE in PDA)
- SEND METHOD

Currently, there is the following limit concerning the usage of large variables:

- CALL statement parameter size less than 64 KB per parameter (no limit for the CALL with INTERFACE4 option).

In the following sections, the use of dynamic variables is discussed in more detail with examples.

Assignments with Dynamic Variables

Generally, an assignment is done in the current used length (*LENGTH) of the source operand.
If the destination operand is a dynamic variable, its current allocated size is possibly extended in order to move the source operand without truncation.

Example:

```
#MyDynText1 := operand or
MOVE operand to #MyDynText1
#MyDynText1 is automatically extended until the source operand can be copied
```

MOVE ALL, MOVE ALL UNTIL with dynamic target operands are defined as follows:

- `MOVE ALL` moves the source operand repeatedly to the target operand until the used length (*LENGTH) of the target operand is reached. *LENGTH is not modified. If *LENGTH is zero, the statement will be ignored.
- `MOVE ALL operand1 TO operand2 UNTIL operand3` moves operand1 repeatedly to operand2 until the length specified in operand3 is reached. If operand3 is greater than *LENGTH(operand2), operand2 is extended and *LENGTH(operand2) is set to operand3. If operand3 is less than *LENGTH(operand2), the used length is reduced to operand3. If operand 3 equals *LENGTH(operand2), the behavior is equivalent to MOVE ALL.

Example:

```
#MyDynText1 := 'ABCDEFGHIJKLMNO'          /* *LENGTH(#MyDynText1) is 15
MOVE ALL 'AB' TO #MyDynText1              /* content of #MyDynText1 is 'ABABABABABABABA';
                                          /* *LENGTH is still 15
MOVE ALL 'CD' TO #MyDynText1 UNTIL 6      /* content of #MyDynText1 is 'CDCDCD';
                                          /* *LENGTH is 6
MOVE ALL 'EF' TO #MyDynText1 UNTIL 10     /* content of #MyDynText1 is 'EFEFEFEFEF';
                                          /* *LENGTH is 10
```

MOVE JUSTIFIED is rejected at compile time if the target operand is a dynamic variable.

MOVE SUBSTR and MOVE TO SUBSTR are allowed. MOVE SUBSTR will lead to a runtime error if a sub-string behind the used length of a dynamic variable (*LENGTH) is referenced. MOVE TO SUBSTR will lead to a runtime error if a sub-string position behind *LENGTH + 1 is referenced, because this would lead to an undefined gap in the content of the dynamic variable. If the target operand should be extended by MOVE TO SUBSTR (for example if the second operand is set to *LENGTH+1), the third operand is mandatory.

Example:

```
/* valid
#op2 := *LENGTH(#MyDynText1)
MOVE SUBSTR (#MyDynText1, #op2) TO operand               /* move last character to operand
#op2 := *LENGTH(#MyDynText1) + 1
MOVE operand TO SUBSTR(#MyDynText1, #op2, #len_operand)
                                                         /* concatenate operand to #MyDynText1
/* invalid
#op2 := *LENGTH(#MyDynText1) +1
MOVE SUBSTR (#MyDynText1, #op2, 10) TO operand           /* leads to runtime error; undefined sub-string
#op2 := *LENGTH(#MyDynText1 +10)
MOVE operand TO SUBSTR(#MyDynText1, #op2, #len_operand)
       /* leads to runtime error; undefined gap
#op2 := *LENGTH(#MyDynText1) +1
MOVE operand TO SUBSTR(#MyDynText1, #op2)                /* leads to runtime error; undefined length
```

Assignment Compatibility

Example:

```
#MyDynText1   := #MyStaticVar1
#MyStaticVar1 := #MyDynText2
```


If the source operand is a static variable, the used length of the dynamic destination operand (*LENGTH(#MyDynText1)) is set to the format length of the static variable and the source operand is copied in this length including trailing blanks (Format A) or zeros (Format B).

If the destination operand is static and the source operand is dynamic, the dynamic variable is copied in its currently used size. If this size is less than the format length of the static variable, the rest is filled with blanks or zeros. Otherwise, the value will be truncated. If the currently used size of the dynamic variable is 0, the static target operand is filled with blanks or zeros.

Initialization of Dynamic Variables

Dynamic Variables can be initialized with blanks (alphanumeric) or zeros (binary) up to the currently used length (= *LENGTH) using the RESET statement. *LENGTH is not modified.

Example:

```
DEFINE DATA LOCAL
:
#MyDynText1   (A)  DYNAMIC
:
END-DEFINE
:
#MyDynText1 := 'short text'
write *LENGTH(#MyDynText1)          /* used length is 10
:
RESET #MyDynText1                   /* used length is still 10, value is 10 blanks
:
```

To initialize a dynamic variable with a specified value in a specified size, the MOVE ALL UNTIL statement may be used.

Example:

```
:
MOVE ALL 'Y' TO #MyDynText1 UNTIL 15          /* #MyDynText1 contains 15 'Y's, used length is 15
:
```

String Manipulation with Dynamic Alpha Variables

If a modifiable operand is a dynamic variable, its current allocated size is possibly extended in order to perform the operation without truncation or an error message. This is valid for the concatenation (COMPRESS) and separation of dynamic alphanumeric variables (SEPARATE).

Example:

```
DEFINE DATA LOCAL
   1  #MyDynText1          (A)  DYNAMIC
   1  #MyDynText2          (A)  DYNAMIC
...
COMPRESS ... INTO #MyDynText2

#MyDynText2 will be extended in order to compress the source operands.
Note: in case of non-dynamic variables the value may be truncated.

SEPARATE INTO #MyDynText1 #MyDynText2 WITH DELIMITER
#MyDynText1 and #MyDynText2 are possibly extended or reduced to separate the source operand.

EXAMINE #MyDynText1 FOR REPLACE
#MyDyntext1 will possibly be extended or reduced to perform the REPLACE operation successfully.
```

Note: In case of non-dynamic variables, an error message may be returned.

Logical Condition Criterion (LCC) with Dynamic Variables

Generally, a read-only operation (such as LCC) with a dynamic variable is done with its currently used size. Dynamic variables are processed like static variables if they are used in a read-only (non-modifiable) context.

Example:

```
IF  #MyDynText1  =  #MyDynText2  OR  #MyDynText1 =  "**"
IF  #MyDynText1  <  #MyDynText2  OR  #MyDynText1 <  "**"
IF  #MyDynText1  >  #MyDynText2  OR  #MyDynText1 >  "**"
```

Also in the case of trailing blanks or zeros, dynamic variables will show an equivalent behavior.
For dynamic variables, the alphanumeric value ’AA ’ will be equal to ’AA’ and the binary value ’00003031’ is equal to ’3031’. If a comparison result should only be TRUE in case of an exact copy, the used lengths of the dynamic variables have to be compared in addition. If one variable is an exact copy of the other, their used lengths are also equal.

Example:

```
#MyDynText1  :=  ’hello’          /* used length is 5
#MyDynText2  :=  ’hello     ’     /* used length is 10
IF  #MyDynText1  =  #MyDynText2   /* TRUE
:
IF   #MyDynText1  =  #MyDynText2
  AND  *LENGTH(#MyDynText1) = *LENGTH(#MyDynText2)
                                  /* FALSE
:
```

Two dynamic variables are compared position by position from left to right up to the minimum of their used lengths. The first position where the variables are not equal determines if the first or the second variable is greater than, less than or equal to the other. The variables are equal if they are equal up to the minimum of their used lengths and the rest of the longer variable contains only blanks (Format A) or zeros (Format B).

Example:

```
#MyDynText1  :=  ’hello1’                /* used length is 6
#MyDynText2  :=  ’hello2’                /* used length is 10
IF  #MyDynText1   #MyDynText2            /* TRUE : #MyDynText2
:="hallo" IF #MyDynText1>  #MyDynText2   /* TRUE
:
```

Comparison Compatibility

Comparisons between dynamic and static variables are equivalent to comparisons between dynamic variables. The format length of the static variable is interpreted as its used length.

Example:

```
#MyStatText1 :=  'hello'           /* format length of MyStatText1 is 20
#MyDynText1  :=  'hello'           /* used length is 5
IF  #MyStatText1  =  #MyDynText1   /* TRUE
:
IF  #MyStatText1  >  #MyDynText1   /* FALSE
```


Parameter Transfer with Dynamic Variables

Dynamic variables may be passed as parameters to a called program object (CALLNAT, PERFORM).
Call-by-reference is possible because the value space of a dynamic variable is contiguous. Call-by-value causes an assignment with the variable definition of the caller as the source operand and the parameter definition as the destination operand. Call-by-value result causes in addition the movement in the opposite direction.
For call-by-reference, both definitions must be DYNAMIC. If only one of them is DYNAMIC, a runtime error is raised. In the case of call-by-value (result), all combinations are possible. The following table illustrates the valid combinations:

Call By Reference

	Parameter	
Caller	**Static**	**Dynamic**
Static	Yes	No
Dynamic	No	Yes

The formats of dynamic variables A or B must match.

Call by Value (Result)

	Parameter	
Caller	**Static**	**Dynamic**
Static	Yes	Yes
Dynamic	Yes	Yes

Note:
In the case of static/dynamic or dynamic/static definitions, a value truncation may occur according to the data transfer rules of the appropriate assignments.

Example 1:

```
DEFINE DATA LOCAL
    1 #MyText (A) DYNAMIC
:
#MyText :=  '123456'           /* extended to 6 bytes
WRITE *LENGTH(#MyText)         /* is 6
CALLNAT  'SUB1' USING #MyText
WRITE *LENGTH(#MyText)         /* is 8; allocated size is 8


Subpgm SUB1:
DEFINE DATA PARAMETER
    1 #MyParm (A)  DYNAMIC BY VALUE RESULT
:
WRITE *LENGTH(#MyParm)     /*is 6; temporary value space of 6 bytes is allocated for #MyParm
#MyParm := '1234567'       /* extended to 7
#MyParm := '12345678'      /* allocated size=8 bytes
EXPAND DYNAMIC VARIABLE #MyParam TO 10
WRITE *LENGTH(#MyParm)     /* is 8; allocated size is 10
END                        /* contents of  #Myparm is moved (back) to #MyText
                           /* used length is 8; #MyText is extended to 8
```


Example 2:

```
DEFINE DATA LOCAL
    1 #MyText (A) DYNAMIC
:
#MyText :=  '123456'                  /* extended to 6 bytes
WRITE *LENGTH(#MyText)                /* is 6
CALLNAT  'SUB2' USING #MyText
WRITE *LENGTH(#MyText)                /* is 8; allocated size is 10 (extended in SUB2)


Subpgm SUB2:
DEFINE DATA PARAMETER
    1 #MyParm (A)  DYNAMIC
:
WRITE *LENGTH(#MyParm)                /* is 6
#MyParm := '1234567'                  /* used length is 7
#MyParm := '12345678'                 /* extended to 8 bytes
EXPAND DYNAMIC VARIABLE #MyParm TO 10
WRITE *LENGTH(#MyParm)                /* is 8; allocated size is 10
END
```


CALL 3GL Program

Dynamic and large variables can sensibly be used with the CALL statement when the option INTERFACE4 is used. The usage of this option leads to an interface to the 3GL program with a different parameter structure. This usage requires some minor changes in the 3GL program, but provides the following significant benefits as compared with the older FINFO structure. For further information on the FINFO structure, see the Call Interface4 statement.

- No limitation on the number of passed parameters (former limit 40).
- No limitation on the parameter's data size (former limit 64 KB per parameter).
- Full parameter information can be passed to the 3GL program including array information.
  Exported functions are provided which allow secure access to the parameter data (formerly you had to take care not to overwrite memory inside of Natural)

Before calling a 3GL program with dynamic parameters, it is important to ensure that the necessary buffer size is allocated. This can be done explicitly with the EXPAND statement.
If an initialized buffer is required, the dynamic variable can be set to the initial value and to the necessary size by using the MOVE ALL UNTIL statement. Natural provides a set of functions that allow the 3GL program to obtain information about the dynamic parameter and to modify the length when parameter data is passed back.

Example:

```
MOVE ALL ' ' TO  #MyDynText1 UNTIL 10000      /* a buffer of length 10000 is allocated
                                              /* #MyDynText1 is initialized with blanks
                                              /* *LENGTH is set is set to 10000
CALL INTERFACE4  #3GLprog  USING  #MyDynText1
write *LENGTH(#MyDynText1)                    /* used length may have changed in the 3GL program
:
```

For a more detailed description, refer to the CALL statement in the Statements documentation.

Work File Access with Large and Dynamic Variables

Large and Dynamic Variables can be written into work files or read from work files using the two work file types PORTABLE and UNFORMATTED. For these types, there are no size restrictions for large/dynamic variables.

The other work file types (ASCII, ASCII-COMPRESSED, ENTIRECONNECTION, SAG and TRANSFER) cannot handle dynamic variables and will produce an error. Large variables for these work file types pose no problem unless the maximum field/record length is exceeded (Field Length 255 for ENTIRECONNECTION and TRANSFER, Record Length 32767 for the others).

For the work file type PORTABLE, the field information is stored within the work file. The dynamic variables are resized during READ if the field size in the record is different from the current size.

The work file type UNFORMATTED can be used, for example, to read a video from a database and store it in a file directly playable by other utilities. In the WRITE WORK statement, the fields are written to the file with their byte length. All data types (DYNAMIC or not) are treated the same. No structural information is inserted. Note that Natural uses a buffering mechanism, so you can expect the data to be completely written only after a CLOSE WORK. This is especially important if the file is to be processed with another utility while Natural is running.

With the READ WORK statement, fields of fixed length are read with their whole length. If the end-of-file is reached, the rest of the current field is filled with blanks. The following fields are unchanged.
In the case of DYNAMIC data types, all the rest of the file is read unless it exceeds 1 GB. If the end of file is reached, the remaining fields (variables) are kept unchanged (normal Natural behavior).

DDM Generation and Editing for Varying Length Columns

Depending on the data types, the related database format A or format B is generated. For the databases' data type VARCHAR the NATURAL length of the column is set to the maximum length of the data type as defined in the DBMS. In case of large data types the keyword DYNAMIC is generated at the length field position.

For all varying length columns, an LINDICATOR field L@<column-name> will be generated. For the databases' data type VARCHAR, an LINDICATOR field with Format/Length I2 will be generated. For large data types (see list below) the format/length will be I4.

In the context of database access, the LINDICATOR handling offers the chance to get the size of the field to be read or to set the size of the field to be written independent of a defined buffer length (or independent of *LENGTH). Usually, after a retrieval function, *LENGTH will be set to the corresponding length indicator value.

Example DDM:

```
 T  L  Name                     F   Leng        S   D   Remark
    :
    1  L@ PICTURE1              I   4                                  /* length indicator
    1  PICTURE1                 B   DYNAMIC             IMAGE
    1  N@PICTURE1               I   2                                  /* NULL indicator
    1  L@TEXT1                  I   4                                  /* length indicator
    1  TEXT1                    A   DYNAMIC             TEXT
    1  N@TEXT1                  I   2                                  /* NULL indicator
    1  L@DESCRIPTION            I   2                                  /* length indicator
    1  DESCRIPTION              A   1000                VARCHAR(1000)
    :
    :
 ~~~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/* concerning PICTURE1
  Header               :    ---
  Edit Mask            :    ---
  Remarks              :   IMAGE
```

The generated formats are varying length formats. The Natural programmer has the chance to change the definition from DYNAMIC to a fixed length definition (extended field editing) and can change, for example, the corresponding DDM field definition for VARCHAR data types to a multiple value field (old generation).

Example:

```
 T  L  Name                     F   Leng          S   D   Remark
    :
    1  L@ PICTURE1              I   4                                  /* length indicator
    1  PICTURE1                 B   1000000000            IMAGE
    1  N@PICTURE1               I   2                                  /* NULL indicator
    1  L@TEXT1                  I   4                                  /* length indicator
    1  TEXT1                    A   5000                  TEXT
    1  N@TEXT1                  I   2                                  /* NULL indicator
    1  L@DESCRIPTION            I   2                                  /* length indicator
 M  1  DESCRIPTION              A   100                   VARCHAR(1000)
      :
      :
 ~~~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/* concerning PICTURE1
  Header               :    ---
  Edit Mask            :    ---
  Remarks              :   IMAGE
```

Accessing Large Database Objects

To access a database with large objects (CLOBs or BLOBs), a DDM with corresponding large alphanumeric or binary fields is required. If a fixed length is defined and if the database large object does not fit into this field, the large object is truncated. If the programmer does not know the definitive length of the database object, it will make sense to work with dynamic fields. As many reallocations as necessary are done to hold the object. No truncation is performed.

Example Program:

```
DEFINE DATA LOCAL
:
1 person VIEW OF xyz-person
  2 nachname
  2 vorname_1
  2 L@PICTURE1                  /* I4 length indicator for PICTURE1
  2 PICTURE1                    /* defined as dynamic in the DDM
  2 TEXT1                       /* defined as non-dynamic in the DDM
:
END-DEFINE
:
SELECT * INTO VIEW person FROM xyz-person                  /* PICTURE1 will be read completely
                            WHERE nachname = 'SMITH'       /* TEXT1 will be truncated to fixed length 5000
:
   WRITE  'length of PICTURE1: '  L@PICTURE1               /* the L-INDICATOR will contain the length
   :                                                       /* of PICTURE1 (= *LENGTH(PICTURE1)
   /* do something with PICTURE1 and TEXT1
:
   L@PICTURE1  :=  100000
   INSERT INTO xyz-person  (*) VALUES (VIEW person)        /* only the first 100000 Bytes of PICTURE1
   :                                                       /* are inserted
:
END-SELECT
```

If a format-length definition is omitted in the view, this is taken from the DDM.
In reporting mode, it is now possible to specify any length, if the corresponding DDM field is defined as DYNAMIC. The dynamic field will be mapped to a field with a fixed buffer length. The other way round is not possible.

DDM format/length definition	**VIEW format / length definition**	
(An)	-	valid
	(An)	valid
	(Am)	only valid in reporting mode
	(A) DYNAMIC	invalid
(A) DYNAMIC	-	valid
	(A) DYNAMIC	valid
	(An)	only valid in reporting mode
	(Am / i : j)	only valid in reporting mode

(equivalent for Format B variables)

Parameter with LINDICATOR Clause in SQL Statements

If the LINDICATOR field is defined as I2 field, the SQL data type VARCHAR is used for sending or receiving the corresponding column. If the LINDICATOR host variable is specified as I4, a large object data type (CLOB/BLOB) is used.

If the field is defined as DYNAMIC, the column is read in an internal loop up to its real length. The LINDICATOR field and *LENGTH are set to this length. In the case of a fixed-length field, the column is read up to the defined length. In both cases, the field is written up to the value defined in the LINDICATOR field.

Performance Aspects with Dynamic Variables

EXPAND and REDUCE

The amount of the allocated memory of a dynamic variable may be reduced to a specified size using the REDUCE DYNAMIC VARIABLE statement. In order to (re)allocate a variable to a specified size, the EXPAND statement can be used. (If the variable should be initialized, use the MOVE ALL UNTIL statement.)

Example:

```
DEFINE DATA LOCAL
:
#MyDynText1    (A)  DYNAMIC
# len          (I4)
:
END-DEFINE

#MyDynText1 := 'a'                 /* used length is 1, value is 'a'; allocated size is still 1

EXPAND DYNAMIC VARIABLE  #MyDynText1 TO 100
                                   /* used length is still 1, value is 'a'; allocated size is 100

CALLNAT #subprog  USING  #MyDynText1
write *LENGTH(#MyDynText1)         /* used length and allocated size may have changed in the subprogram

#len := *LENGTH(#MyDynText1)
REDUCE DYNAMIC VARIABLE  #MyDynText1  TO  #len
                                   /* if allocated size is greater than used length, the unused memory is released
:
REDUCE DYNAMIC VARIABLE  #MyDynText1  TO  0
                                   /* free allocated memory for dynamic variable
END
```

Rules:

- Use dynamic operands where it makes sense.
- Use EXPAND if upper limit of memory usage is known.
- Use REDUCE if the dynamic operand will no longer be needed.

